Evaluating low-level software-based hardening techniques for configurable GPU architectures

Noname manuscript No.
(will be inserted by the editor)

Evaluating Low-level Software-based Hardening Techniques for Configurable GPU Architectures

Marcio M. Goncalves[1] ·
Josie E. Rodriguez Condia[2] ·
Matteo Sonza Reorda[2] ·
Luca Sterpone[2] ·
Jose Rodrigo Azambuja[1]

Received: date / Accepted: date

Abstract The high processing power of GPUs makes them attractive for safety-critical applications, where transient effects are a major concern, and resilience must be enforced without compromising performance. Configurable softcore GPUs are a recent technology that allows detailed reliability assessment capable of bringing directions to the design of reliable GPU applications. This work investigates the reliability of the register files and the pipeline of a softcore GPU under radiation-induced faults. It proposes software-based fault tolerance techniques to mitigate errors. Faults are simulated at the register transfer level in four case-study algorithms, and the Architectural Vulnerability Factor (AVF) and Mean Workload to Failure (MWTF) are checked over different GPU configurations. Results indicate that software-based techniques efficiently reduce AVF. In terms of MWTF, results show that the best cases depend on an optimized balance between GPU configuration, application runtime, and AVF.

Keywords Fault tolerance, Graphics processing units, Single event upsets, Software-based hardening techniques.

Marcio M. Goncalves
E-mail: mmgoncalves@inf.ufrgs.br

Josie E. Rodriguez Condia
E-mail: josie.rodriguez@polito.it

Matteo Sonza Reorda
E-mail: matteo.sonzareorda@polito.it

Luca Sterpone
E-mail: luca.sterpone@polito.it

Jose Rodrigo Azambuja
E-mail: jose.azambuja@inf.ufrgs.br

[1] Federal University of Rio Grande do Sul (UFRGS), Brazil
[2] Department of Control and Computer Engineering (DAUIN), Politecnico di Torino, Italy

1 Introduction

Graphics Processing Units (GPUs) have been originally designed for graphics applications but soon evolved into general-purpose applications due to their high computing power and the advent of programming support for parallel applications. Over the last decade, the rapid proliferation of GPUs has reached safety-critical applications, such as automotive and aerospace [1], and High-Performance Computing (HPC) applications, such as cloud, radars, and others [2, 3]. NVIDIA GPUs, for instance, are used as accelerators in several top500 supercomputers as well as self-driving cars [4].

Modern GPU architectures are designed with a Reduced Instruction Set Computing (RISC) architecture in a Single Instruction Multiple Data (SIMD) paradigm for exploiting data parallelism. The architecture relies on a single execution pipeline that computes a single instruction flow on all computing units in parallel. To do so, it operates over large blocks of data in parallel. In this sense, most instructions in a kernel work directly on these registers, communicating as little as possible with global and shared memories, thus hardening techniques must act locally to stop kernel faults from spreading to shared and global memories. From a reliability point of view, storage elements allocated in the register file and the pipeline are critical resources, causing permanent or temporary effects on digital systems. Knowing the probability of an error in a register to propagate to the outputs may be sufficient to characterize an application's vulnerability.

Faults on electronic components are mainly caused by energized particles from cosmic rays and high energy protons as transient pulses in logic or support circuitry and can cause permanent or temporary effects over the system [5]. Among the most common non-destructive effects is the Single Event Upset (SEU), also known as a bit-flip. SEU effects play a major role in GPU architectures because they require register files to be large and extremely fast, which inherently makes them more susceptible to the effects of transient faults. The use of Error Correction Codes (ECCs) can mitigate transient faults. Still, the presence of ECC in GPU register files is not mandatory, and its availability varies from device to device. Also, ECCs present drawbacks in reliability, large overheads in area, and even performance degradation [6, 7].

SEU effects tend to increase in cutting-edge semiconductor technologies, where operating frequencies and transistor density are higher and voltage supplies are lower. They grow to the point where the newest GPUs are prone to experience radiation-induced errors [8, 9], even on applications running at ground level, where neutrons are the primary source of soft errors [10]. Also, the clustering of GPUs in large-scale HPC systems increases the fault occurrence frequency down to an order of minutes [11], that value being far higher in aerospace environments. The reliability of GPUs is such an open issue that the increased number of errors are starting to outweigh their performance benefits [12].

In safety-critical and HPC application domains, fault tolerance techniques are mandatory to detect or correct faults. Safety-critical applications cannot

result in erroneous output data, as they can directly impact human lives. Still, they can crash in many cases, as long as we can detect this faulty situation through a watchdog timer. On the other hand, HPC applications can result in erroneous data, but crashing means breaking timing constraints, which is not acceptable. Therefore, both application classes must work adequately despite the existence of faults. However, the reliability of GPUs is still an open issue.

Software-implemented fault tolerance techniques are an alternative to hardening GPU hardware against transient effects. They have been proposed and applied for GPUs in the past years, targeting registers in the registers files and the pipeline [13, 14], with high detection rates and high costs in performance degradation [15]. Unlike hardware-based techniques, software-implemented ones can use Commercial-Off-The-Shelf (COTS) hardware, modifying the software compilation flow to harden applications against SEU effects. The downside is that software modifications require more execution time, therefore decreasing performance. Software transformation tools can automatically apply these techniques to the source code of a program, thus simplifying the task for software developers: by protecting the system by acting on the software, they can reduce development costs significantly [16]. With the recent advent of configurable GPUs [17, 18], engineers became able to design effective fault tolerance techniques better. Still, the literature does not present an experimental analysis on the design space exploration of software-based solutions for GPU architectures, especially for configurable open-source ones.

This paper is an extension of Gonçalves et al. [19] and presents a comprehensive analysis of low-level software-based hardening techniques developed to harden a General Purpose GPU (GPGPU) architecture against SEUs. This work targets an open-source configurable GPU based on the NVIDIA G80 architecture. However, one could extend our proposed analysis and implementation to other GPU architectures, including commercial ones. We also propose three optimizations to reduce execution time while maintaining fault detection capabilities. A fault injection campaign is performed at the Register-Transfer Level (RTL) using FlexGripPlus, a GPGPU based on the NVIDIA G80 architecture, targeting registers in the register file and the pipeline. Techniques and their optimizations are evaluated for three GPU configurations in terms of execution time, Architectural Vulnerability Factor (AVF), and Mean Work to Failure (MWTF). Results show that an analysis is required to navigate the hardened GPU application design space effectively because even though execution time, AVF, and error detection are important factors, they are not enough.

The main contributions of this work are:

- The adaptation of state-of-the-art software-based hardening techniques at the assembly-level to the GPU architecture, and their evaluation on an open-source model;
- Three optimizations for the software-based hardening techniques targeting GPU architectures for more effective fault detection;

– An analysis of how different GPU configurations absorb the execution time overhead imposed by software-based hardening techniques when increasing the number of cores;
– A fault injection campaign at RTL with over 1.4 million faults for the target GPU running four applications on three GPU configurations;
– A comprehensive design space exploration for hardened applications running in multiple GPU configurations considering execution time, AVF, and MWTF.

The remainder of this paper is organized as follows. In Section 2, we discuss related work on software-based transformation techniques and their application to GPU architectures. Section 3 describes the chosen target GPU architecture and its vulnerabilities. Sections 4 and 5 describe the proposed software-based hardening techniques and discuss their implementation, respectively. In Section 6, we present results from the fault injection campaign, assessing the detection capabilities of the proposed techniques. Section 7 introduces the MWTF as an additional reliability metric and presents the reliability design space exploration for multiple configurations. Finally, we close this work in Section 8 with concluding remarks and directions for future research.

2 Related Work

Over the last years, authors have proposed software-based fault tolerance techniques for GPUs in the literature. They have been implemented in a wide range of abstraction levels, from the high-level (i.e., CUDA for NVIDIA GPUs), where threads and variables are replicated, to low-level (i.e., assembly), where instructions and registers are replicated, exploiting naive and partial duplication [20, 21]. The strategies also include Algorithm-Based Fault Tolerance (ABFT) techniques, which can achieve high detection rates at low execution time overheads but are limited to a specific group of applications [22, 23, 24]. More recently, the use of hybrid software-hardware-based techniques and approximate computing has been explored towards the design of resilient GPU applications [15, 25, 26, 27].

At the application level, Dimitrov et al. [20] proposed three techniques that leverage Thread-Level Parallelism (TLP) and Instruction-Level Parallelism (ILP) to replicate the application code, thus mitigating the effects of transient faults in the GPU. Their techniques resulted in performance overheads of up to 100%. Similarly, authors in [6] proposed DWC strategies that explore spatial and temporal redundancy by duplicating blocks and performing a redundant thread execution after the original one. They were able to achieve up to 100% error reduction at the cost of increasing execution time to 2.5 times the original one. Wadden et al. [21] proposed a compiler-based approach for GPUs that converts kernels into redundantly threaded versions and observed high overheads for inter-thread communication and synchronization. The results demonstrated that performance costs depend on the application's workload, reaching more than 100% in some cases. Gupta et al. [28] extended

the work done in [21] by proposing compiler optimizations to reduce the high synchronization overhead of redundant multi-threading on GPUs. Software-based instruction-level duplication does not incur high synchronization overheads because instruction duplication and consistency checking are performed inside each thread.

At the assembly level, authors in [13] proposed fault tolerance techniques to detect faults in the register files of GPUs by replicating assembly instructions in an intertwined fashion. They were able to achieve a 99% error reduction at an increase in execution time of 78%. Authors in [15] introduced a group of software-based fault tolerance techniques to optimize instruction-level duplication for GPUs. The optimizations improved resilience by up to 87%, with an average execution time increase of 36%. More recently, hybrid approaches, including software and hardware modifications, have been proposed to improve software-based techniques. Authors in [25] proposed a cooperative hardware-software mechanism to detect errors in a GPU's pipeline with low instruction-duplication overhead. The authors in [15] proposed ISA extensions to eliminate consistency checks and notification instructions. The authors in [26] proposed reliable atomic memory access and predicate setting instructions to improve performance and fault detection capabilities of software-based fault tolerance techniques. These techniques could lower runtime overhead significantly at low costs in area overhead.

Partial hardening techniques can be adopted at all abstraction levels. They usually decrease performance cost in exchange for less fault coverage by selectively duplicating structures (instructions, threads, registers, etc.) based on their criticalities [27, 29]. Authors in [30] employed heuristics to identify and protect critical instructions and achieved an average reduction of 100% of DUEs and 98% of SDCs by covering 90% of the dynamic instructions. Authors in [31] were able to evaluate register criticality on a commercial GPU, apply selective fault tolerance techniques through software modifications and correlate results with hardware-implemented fault tolerance techniques. This work showed that selective hardening presents better efficiency in terms of fault coverage per overhead than applying random or full register file hardening. For instance, results showed up to 65% SDC coverage by only protecting 30% of the registers. Partial hardening can be further improved by combining approximate computing. In this sense, authors in [27] used approximate computing to rank the application's most critical registers based on the magnitude of the output error that they provoked when a fault corrupted them. When compared to selective hardening techniques, they reduced replicated registers by 41% on average while maintaining SDC fault coverage. Authors in [32] proposed to leverage the concept of approximate computing and mixed-precision architectures to improve DWC's runtime at the cost of less fault coverage by replicating the original execution in a lower precision. Results showed an average fault coverage of 73% at the average cost of 16% in runtime overhead when considering two applications from the HPC domain.

In this work, we propose the adoption of software-based fault-tolerance techniques through low-level assembly code transformations. By operating at

the assembly code, the proposed transformations are independent of the compiler and the high-level programming language. Therefore, they can be applied in tandem with previous works proposed in the literature. Then, we propose three optimizations targeting GPU architectures to either improve or trade-off performance for fault detection. These novel technique optimizations, called *Traceback*, *Move*, and *Delayed Notification*, are used to selectively protect memory access and predicate setting instructions. Moreover, we investigate the sensitivity of register files and pipeline registers to radiation-induced faults. To do so, we use an open-source GPU RTL model [17] to perform a simulation-based fault injection campaign with over 1.4 million faults at RTL with different GPU core configurations. Finally, we perform a comprehensive analysis and exploration of the design space for hardened GPU applications, investigating the impact of the proposed hardening techniques in execution time, AVF, and MWTF.

3 FlexGripPlus Architecture

FlexGripPlus [17] is an open-source configurable GPU softcore model described in VHDL that implements the NVIDIA G80 micro-architecture [33]. This model can be programmed using the CUDA programming environment, with the support of up to 52 assembly instructions (SASS). The GPU model structure is composed of an array of Streaming Multiprocessors (SMs) that execute threads in parallel. Fig. 1 depicts the general organization of one SM core in the FlexGripPlus architecture.

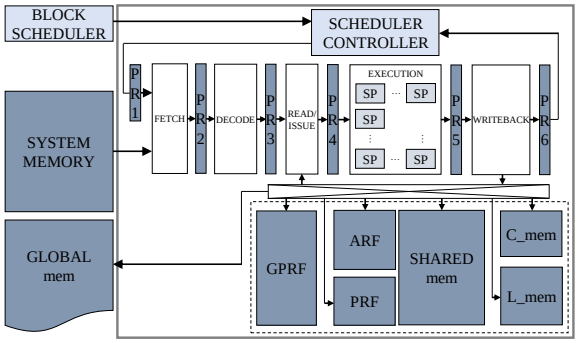


Fig. 1 The general scheme of an SM core in the FlexGripPlus model

Each SM executes instructions in a parallel manner following the Single-Instruction Multiple-Thread (SIMT) paradigm [34]. The SMs are managed by a Block Scheduler Controller, which distributes the workload into each available SM in the system. Internally, the SM is divided into a five-stage

pipeline and includes one Warp Scheduler Controller managing and monitoring the concurrent execution of a group of 32 parallel threads (also known as warp).

Inside the SM, a set of pipeline registers (PRx) are located between two consecutive stages storing data path and control path signals to exploit the ILP. Similarly, the high-performance parallel execution is achieved through a supporting memory hierarchy in the GPU. The memory system is composed of a General-Purpose Register File (GPRF), an Address Register File (ARF), a Predicate Register File (PRF), a local memory (L_mem), a constant memory(C_mem), a shared memory, and the global memory.

On the one hand, some external memory resources, such as the local memory, are mainly employed to store arrays. Furthermore, the constant memory is utilized to store constant values for all threads during a program's execution. The shared memory stores data operands that can be used among threads belonging to the same block. Finally, the global memory stores the initial inputs and the final results of a program kernel. The host computer then retrieves these values.

On the other hand, the GPRF, ARF, and PRF are located inside the SM. A 16KB-size GPRF is the leading and fastest memory resource in the SM. This GPRF is commonly used for every thread to store data operands, addresses, and results during the program execution. The PRF stores predicates as a result of logic-arithmetic or comparison instructions, with up to four predicate registers per thread. These resources (GPRF, ARF, and PRF) are statically organized in banks as the available number of cores (also known as Streaming Processor or (SP)) and can only be accessed by the associated SP. The number of registers per thread in the GPRF directly depends on the application and the total number of active threads, attaining a maximum of 64 registers per thread on each bank.

FlexGripPlus can be configured to operate with 8, 16, or 32 cores inside each SM. This allows to flexibly modify the length of the data path proportionally in the pipeline registers and the register size per core in each bank of the GPRF to 2KB, 1KB, and 512B for the previously specified core configuration, respectively. The fixed size of the whole GPRF implies that the register bank size is defined when selecting the number of execution cores per SM.

4 Software-based Hardening Techniques

This Section discusses the main program code transformations for applying software-based hardening techniques based on instruction and data duplication and the three proposed optimizations to improve performance at different costs in reliability, fault effects, and fault notification time.

The software-based hardening techniques discussed in this paper are implemented through program code transformations at the assembly level. Thus, they insert/remove assembly instructions into/from the program code, access registers and memory addresses, and use the datapath and the controlpath through assembly instructions. The main benefit of applying software-based

techniques at the assembly level is that they are compiler-independent. Therefore, all compiler optimizations can be performed without removing the added redundancies, and we have better control over the code transformation. Also, one can directly target specific registers instead of variables, directly protecting the register files. The main drawback is that the pipeline is not directly accessible. Therefore, its hardening becomes a byproduct of the register file hardening.

To achieve the desired software-based hardening, we present three code transformations at the assembly level based on the literature [35] and propose three optimizations to decrease the performance degradation (i.e., increase their performance) of the given transformations. In the following, we discuss the program code transformations and the proposed optimizations in detail.

4.1 Program Code Transformations

The three program code transformations discussed in this work are based on the original work of Oh et al. [35] for CPUs and focus on duplicating the datapath operations, checking the results for consistency, and notifying the host computer in case of divergence. To duplicate the datapath operations, we first have to duplicate the registers used by the datapath. In order to do so, we first perform static code analysis to find which registers are being used by the target application and which are not (*spare registers*). We then create a hash table, assigning a spare register to each used register as a copy register. In case there are not enough spare registers to fully duplicate the datapath, one must either select a group of registers and perform selective hardening, thus decreasing reliability, or perform register spilling into the memory, thus decreasing the application's performance. All our case-study applications had enough spare registers for full datapath duplication. Once static registers assignment is completed, we perform the following three transformations: (1) datapath duplication, (2) consistency checking, and (3) host notification.

Datapath duplication (transformation 1 - T1) is responsible for duplicating all datapath instructions. It is the main software-based transformation. It forces the hardware to execute twice the datapath operation in an intertwined fashion, exploiting Instruction Level Parallelism (ILP) from the GPU architecture more effectively than running the code twice. The replicated instructions also perform operations on the copy registers, completely separating the original and duplicated datapath operations. As we consider the memory to be hardened by other means (e.g., ECC), we do not duplicate store instructions, thus also not duplicating memory addresses.

Consistency checking (transformation 2 - T2) is responsible for checking the consistency of registers and their copies. It uses a comparison instruction followed by a conditional branch instruction to an error subroutine. The main issue with checking register consistency is that it creates a dependency between both data flows, unifying them for the comparison instruction, thus decreasing ILP gain. This work evaluates the insertion of consistency checking after two

classes of instructions: memory access and predicate setting. The first affects the data being processed (i.e., data flow), while the latter affects the program's execution (i.e., control flow).

Host notification (transformation 3 - T3) notifies the host of an error. It is usually a trap instruction (exception signal to the host usually seen in GPU ISAs, such as in CUDA) but could also be a memory write instruction signaling the host through the global memory. These instructions are not executed on correct application execution and should be executed a single time when a fault is detected. On the other hand, their execution is conditional to T2, so a predicate register must be checked every time a host notification is added to the program code.

Fig. 2 exemplifies the three transformations in column *Non-optimized Hardened Code*. Datapath duplication (T1) is depicted in green, replicating lines 1, 3, and 5 with lines 2, 4, and 6. As one can notice, the instructions are the same, but they operate over replicated registers (e.g., R3' instead of R3). Note that the store instruction in line 15 is not duplicated. Consistency checking (T2) is highlighted in blue through instructions 7, 16, and 18, which check consistency for memory access instructions, and instructions 10 and 12, which perform consistency checks for predicate setting instructions. Finally, host notification transformation (T3) is shown in red and inserted after each consistency checking instruction in lines 8, 11, 13, 18, and 19.

	Unhardened Code	Non-optimized Hardened Code	Optimized Hardened Code			
			Move	Traceback MEM	Traceback PRED	Delayed Notification
1:	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4	MOV R3, 4
2:		MOV R3', 4;	MOV R3', 4;		MOV R3', 4;	MOV R3', 4;
3:	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;	ADD R1, R1, 1;
4:		ADD R1', R1', 1;	ADD R1', R1', 1;	ADD R1', R1', 1;		ADD R1', R1', 1;
5:	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];	LOAD R2, [R1];
6:		LOAD R2', [R1'];	MOV R2', R2;	LOAD R2', [R1'];		LOAD R2', [R1'];
7:		SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';		@!PE SETP.NE PE, R1, R1';
8:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
9:	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;	SETP.NE P0, R3, R0;
10:		SETP.NE PE, R3, R3';	SETP.NE PE, R3, R3';		SETP.NE PE, R3, R3';	@!PE SETP.NE PE, R2, R2';
11:		@PE ERROR;	@PE ERROR;		@PE ERROR;	
12:		SETP.NE PE, R0, R0';	SETP.NE PE, R0, R0';		SETP.NE PE, R0, R0';	@!PE SETP.NE PE, R3, R3';
13:		@PE ERROR;	@PE ERROR;		@PE ERROR;	
14:	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;	@P0 BRA 1;
15:	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;	STORE [R4], R1;
16:		SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';	SETP.NE PE, R1, R1';		@!PE SETP.NE PE, R1, R1';
17:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
18:		SETP.NE PE, R4, R4';	SETP.NE PE, R4, R4';	SETP.NE PE, R4, R4';		@!PE SETP.NE PE, R4, R4';
19:		@PE ERROR;	@PE ERROR;	@PE ERROR;		
20:						@PE ERROR;

Fig. 2 Software-based hardening technique transformation examples.

4.2 Proposed Optimizations

The presented program code transformations take advantage of ILP but still duplicate the whole datapath (except for store instructions) and insert consistency checks and host notifications. Therefore, we expect them to incur high execution time overheads, even considering ILP gains. To reduce performance degradation (i.e., increase the performance of software-based hardening

techniques), we propose three optimizations targeting the program code transformations: Move optimization, Traceback optimization, and Delayed Notification optimization. These optimizations aim at improving performance by trading-off reliability, detection of specific effects, and host notification delay.

4.2.1 Move optimization

Memory access instructions (i.e., load and store) are the instructions that require the most clock cycles to be executed in FlexGripPlus (e.g., a load instruction requires around four times more clock cycles than a move instruction). In this sense, we propose the *Move* optimization. This optimization replaces replicated load instructions with move instructions that copy the loaded data to the copy register. Thus, instead of adding a replicated load instruction, it adds a faster move instruction. By doing so, it directly affects the datapath duplication (T1) by reducing the performance overhead and adding a point of failure to the hardened code.

Fig. 2 shows an example of this optimization in column *Move opt.* When compared to the non-optimized hardened version (*Hardened code*), the only difference is line 6, where *Hardened code* replicates the original load instruction in line 5 (LOAD R2, [R1]) with a second load instruction (LOAD R2', [R1']), and the Move optimization uses a move instruction instead (MOV R2', R2).

While replicating a load instruction with a move instruction is able to reduce execution time, it inserts a point of failure on the software-based hardening technique, thus trading-off on reliability. Suppose a fault affects the register written by the load instruction before the move instruction can copy its value to the replicated registers. In that case, the corrupted value will propagate to the replicated register, both value and its replica will be corrupted, and the consistency checking will not signal a fault. It is important to notice that, even though the load instruction takes an increased amount of clock cycles to execute, only a fault that hits the instruction in its late write-to-register stage would actually upset the destination register. Therefore, this point of failure is not as large as the fetch-to-fetch time.

4.2.2 Traceback optimization

Datapath duplication and consistency checking are the leading performance degradation causes in software-based hardening techniques, even when considering ILP. One alternative is to selectively apply selective hardening techniques by targeting only the most critical parts of the program code. However, related works still cannot pinpoint the most critical parts of a program code. Some related works target subroutines and functions at the program level [REF], while others target variables, registers, and memory addresses. Instead, we propose the Traceback optimization to target instructions and their data in a more fine-grained approach. It targets specific instructions and all data used during their execution. To do so, we choose a group of target instructions with a high probability of causing an error to the application if affected by a fault

(i.e., memory access instructions for data-flow errors and predicate setting instructions for control-flow errors) and evaluate all previous instructions that lead to the execution of these target instructions (i.e., all instructions that computed the data read by a given target instruction).

To implement the Traceback optimization, we start by defining a group of instructions as target instructions (e.g., memory access or predicate setting instructions). Then, we analyze the static program code and draw its control flow. Next, for each target instruction i, we evaluate which instructions have written its registers and add them to the group of target instructions. We run this procedure recursively until there are no new instructions to be added to the target instruction group. Finally, we apply the previously discussed program code transformations selectively to this group of instructions.

The choice of which instructions to add to the target instruction group is a complex task that depends on in-depth code analysis, fault injection campaigns, and design space exploration. As this analysis is out of the scope of this work, we took a simplified approach and selected two instruction groups: memory access instructions (Traceback MEM) and predicate setting instructions (Traceback PRED). By doing so, we expect to target data-flow and control-flow errors, respectively. Even though this is a simplification of the problem, we expect it to provide us results good enough for a proof-of-concept.

Fig. 2 shows the Traceback optimization applied for memory access instructions on the *Traceback MEM* column and for predicate setting instructions on the *Traceback PRED* column. The original code has two memory access instructions in lines 5 and 15, which have their registers R1 written by instruction 3, and one predicate setting instruction in line 9, which has its register R3 written by instruction 1. Thus, the Traceback optimization selectively hardens lines 3, 5, and 15 for the memory access instructions and lines 1 and 9 for the predicate setting one.

The Traceback optimization targets specific instructions to increase the effectiveness of the program code transformation. In doing so, it leaves parts of the program code unhardened. However, it allows designers to target specific fault effects such as data-flow or control-flow errors. For an application with separated logics for processing its control flow and its data flow (e.g., a constant loop that computes a given value), this optimization can target control-flow and data-flow effects very effectively. However, hardening an application with an entangled logic for computing its control and data flow (e.g., a computation that defines a loop iteration based on inputs) has a higher chance of resulting in the same hardened code as without this optimization.

4.2.3 Delayed Notification optimization

The host notification program code transformation informs the user that a fault has been detected in a previous consistency check. When applying the non-optimized program hardening, the consistency check compares two values and overwrites a predicate register with a flag indicating if a fault was detected. As every consistency check overwrites the previous value of the flag, the host

notification has to be done before the next consistency check. The proposed Delayed Notification optimization changes how consistency checks write to predicate registers and thus provides the designer with options to perform host notifications less frequently. To do so, it employs conditional instructions (usually implemented but not restricted to GPU ISAs, such as in CUDA) to replace the comparison instruction with a conditional comparison instruction. By doing so, the predicate register is only written once when changing state to "fault detected," thus never being reset. Therefore, multiple consistency checks can be paired with a single host notification, up to using a single host notification instruction for the complete program code.

Fig. 2 shows the Delayed Notification optimization in the *Delayed Notification opt.* column. Compared to the non-optimized program hardening, it replaces all consistency checks (SETP.NE) in lines 7, 10, 12, 16, and 18 with conditional consistency checks (@!PE SETP.NE). By doing so, it removes all host notification instructions in lines 8, 11, 13, 17, and 18 and inserts a new host notification instruction in line 20.

The Delayed Notification optimization provides designers with the option to decrease host notification frequency. In doing so, it improves performance at the cost of a larger fault notification period. Therefore, it does not decrease reliability in terms of fault detection. However, a latent fault in the system might increase the chance of fault causing an error. Also, the longer the system takes to inform the host of a fault, the longer the user will take to correct the fault.

5 Application Hardening

The chosen case-study applications are simple but representative when considering resource usage and execution flow orientation: vector sum (VectorSum), matrix multiplication (Matrix), Fast Fourier Transform (FFT), and bitonic sort (Sort). The VectorSum is the shortest application because it only sums two vectors in the memory, therefore being a pure data flow-oriented application. The Matrix is mostly a data flow-oriented application. Still, it has a small fixed loop that iterates over the matrices. The FFT is a mix of control-flow and data-flow orientation. It has a more complex control than the Matrix but still performs heavy multiplications and additions. Finally, the Sort is mostly control-flow oriented, as it moves data according to data comparisons. Even though micro-benchmarks, these applications make the building blocks of major HPC and safety-critical applications [1, 2, 3]. Table 1 shows the resource usage for the four case-study applications running on the 8-, 16-, and 32- core configurations.

The case-study algorithms were implemented in CUDA and compiled with the NVIDIA NVCC compiler. The compilation process generates the CUDA binary file (cubin) containing the assembly code that the NVIDIA GPU modeled by FlexGripPlus effectively executes. This assembly code is called Shader Assembly (SASS) and can be extracted from the cubin file with the cuobjdump

Table 1 Program memory and runtime requirements for all FlexGripPlus configurations

Application	Program Memory (bytes)	Runtime (μs)		
		8 cores	16 cores	32 cores
FFT	1,344	964	588	406
Matrix	264	320	224	177
Sort	288	824	610	502
VectorSum	563	141	103	85
Average	615	562	381	292

tool provided by NVIDIA's CUDA toolkit. We used a tool called HPCT [36], which we upgraded to support the FlexGripPlus ISA and the proposed techniques, to apply the software-implemented techniques to the case-study applications automatically. We input the SASS code to HPCT, automatically applying the code transformations and generating a hardened SASS file.

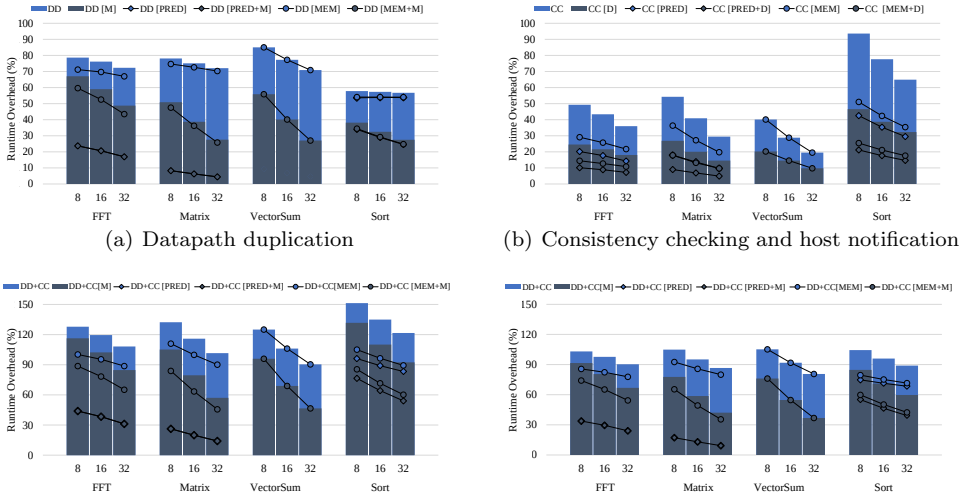
Table 2 shows the percentage execution time overhead over Table 1, individually, for the datapath duplication (T1) - DD - and the consistency checking (T2) with host notification (T3) - CC. Data have been calculated by measuring datapath duplication alone (T1) and its difference to all transformations combined (T1, T2, and T3), thus considering ILP and architectural characteristics of the GPU configuration. Therefore, to effectively harden a program code (and assess execution time overhead), one must combine the datapath duplication column with the consistency checking and host notification column, adding their respective percentage overheads. The datapath duplication column considers data for memory access and predicate setting instruction duplication (DD), Traceback optimization for memory access (DD [MEM]) and predicate setting (DD [PRED]), and the Move optimization ([M]). The consistency checking and host notification column considers data for the Traceback optimization for memory access (CC [MEM]) and predicate setting (CC [PRED]) and the Delayed Notification optimization ([D]). Data for checking both memory access and host notification requires adding columns MEM and PRED (optionally with [D]).

Fig. 3 draws and discusses data from Table 2, considering the execution time overhead for all transformations running on 8-, 16-, and 32-core GPU configurations. Fig. 3(a) depicts isolated overheads for datapath duplication (DD). It shows the following datapath duplication versions: DD, non-optimized; DD [M], optimized with Move; DD [PRED], optimized with Traceback PRED; DD [PRED+M], optimized with Traceback PRED and Move; DD [MEM], optimized with Traceback MEM; and DD [MEM+M], optimized with Traceback MEM and Move. Fig. 3(b) shows the same hardening versions, but for consistency checking and host notification (CC). Finally, Figs. 3(c) and 3(d) shows hardening versions for all transformations (DD+CC) without and with the Delayed Notification optimization, respectively.

Fig. 3(a) shows that the Move optimization produces a significant reduction in the datapath duplication overhead for all applications. This result was

Table 2 Execution time overhead for datapath duplication, consistency checking, and host notification (%)

Application	Cores	Datapath duplication (DD)						Consistency checking and host notification (CC)			
		Full	Full [M]	MEM	MEM [M]	PRED	PRED [M]	MEM	MEM [D]	PRED	PRED [D]
FFT	8	79	67	71	60	24	24	29	14	20	10
	16	76	59	70	53	21	21	26	13	18	9
	32	72	49	67	43	17	17	22	11	14	7
Matrix	8	78	51	75	48	8	8	36	18	18	9
	16	75	39	73	36	6	6	27	13	14	7
	32	72	28	70	26	4	4	20	10	10	5
Sort	8	58	38	54	34	54	34	51	25	43	21
	16	57	32	54	29	54	29	42	21	35	18
	32	57	27	54	25	54	25	35	18	30	15
VectorSum	8	85	56	85	56	9	9	40	20	-	-
	16	77	40	77	40	7	7	29	15	-	-
	32	71	27	71	27	5	5	19	10	-	-
Average	8	75	53	71	50	24	19	39	19	27	13
	16	71	43	69	40	22	16	31	16	22	11
	32	68	33	66	30	20	13	24	12	18	9



(a) Datapath duplication

(b) Consistency checking and host notification

(c) Transformations without Delayed Notification

(d) Transformations with Delayed Notification

Fig. 3 Transformations' runtime overhead for 8-, 16-, and 32-core configurations.

expected, as we designed this optimization to reduce execution time at a cost in reliability, later evaluated in Section 6. On the other hand, the Move optimization also produces an increased reduction for configurations with more cores. This effect can be explained by the fact that an arrangement with more parallel cores increases the number of concurrent global memory accesses. Thus, by replacing load instructions with move instructions, we could alleviate the global memory pressure, which is then able to reduce performance degradation more efficiently in configurations with more cores. These data indicate that the Move optimization can perform even better in COTS GPUs due to their configuration with many cores.

Fig. 3(a) also shows the Traceback optimization applied to the case-study applications. To evaluate this optimization, we have to consider its application to the memory access and predicate setting instructions individually. When the Traceback optimization is applied to the memory access instructions, the performance gain is minimal. This result happens due to two main factors: most of the program code instructions are used for accessing memory data (i.e., data-flow), thus cannot be removed from the hardening; or the data-flow also uses most of the instructions used for calculating conditional branches, conditional executions, and loops (i.e., control-flow). With a small control-flow, data-flow-oriented applications FFT, Matrix, and VectorSum showed little performance improvement. The VectorSum presented no improvement due to not having control-flow instructions. With a complex control flow, the Sort application also showed little performance improvement because of its entangled data- and control-flows. Unlike the Move optimization, the Traceback optimization is not a trade-off between performance and reliability. Therefore, even with small performance gains, there are no drawbacks to applying it.

When considering the Traceback optimization applied to the predicate setting instructions, the performance gain is much higher than when applied to the memory access instructions. For the same reasons discussed previously, applications with a heavy data flow can be more aggressively optimized as long as the data and control flow are not entangled. Therefore, one can notice a significant performance gain for the FFT and Matrix applications. Because the VectorSum has no control flow, its optimization would equal the original unhardened application. On the other hand, the Sort application showed the smallest performance gain because its instructions are mostly used for both control- and data-flow. The optimizations in runtime observed when protecting the FFT and Matrix applications show that the Traceback optimization can harden instructions selectively with significant performance gains.

Fig. 3(b) shows that, for all applications, the Delayed Notification optimization reduces the runtime overhead by half. This reduction happens because it uses a single notification instruction for a group of checking instructions instead of one notification for each checking instruction. The highest reduction is achieved with a single notification instruction for the complete program code. The main drawback is that this optimization increases the average delay between identifying a fault and notifying the host. However, it should not decrease reliability. The selective protection improves performance significantly, especially when associated with the Delayed Notification. The number of cores demonstrates an impact similar to that observed in the datapath replication with the Move optimization (Fig. 3(a)).

The total costs for the proposed fault tolerance techniques are presented in Figs. 3(c) and 3(d), without and with the Delayed Notification optimization, respectively. They add the datapath duplication costs presented in Fig. 3(a) with the consistency checking and host notification costs presented in 3(b). Results without the optimization show that the Move and Delay optimizations together reduce the average performance cost from 134%, 119%, and 105% to 83%, 66%, and 51% for 8, 16, and 32 cores, respectively. When selective mem-

ory protection is implemented, the average cost drops to 69%, 55%, and 42% for 8, 16, and 32 cores. When selective predicate instruction is implemented, the average cost is 35%, 30%, and 24% for 8, 16, and 32 cores, respectively. Such results indicate that, when performance is important, selective protection should be applied whenever possible. In addition, the results show that the amount of cores not only speeds up the execution time of applications but also absorbs the cost of performance of fault tolerance techniques.

6 Fault injection results

We performed extensive fault injection campaigns through simulation at RTL to measure the impact of software-based hardening techniques on reliability. We measured reliability by evaluating the probability of a low-level corruption corresponding to a bit-flip in the register files and the pipeline registers to propagate to the output vector and cause an error. This metric is also known as the AVF [37] and is computed by dividing the number of errors by the number of injected faults. We did not inject faults in the memories, as we assumed them to be protected by design (e.g., ECC), but we intend to evaluate them in the future. To evaluate the system's reliability, we classified the injected faults according to their effect on the system [37]: *Masked*, when the result is correct; *Detected Unrecoverable Error* (DUE), when the application crashes or hangs; *Silent Data Corruption* (SDC), when the program finishes with an incorrect result; and *Detected*, when our software-based hardening techniques detected a fault.

Faults were injected in the original and hardened case-study applications. For each application, we considered three hardened versions: (1) *SDC Hard*, with consistency checking for memory access instructions and the Traceback MEM optimization, (2) *DUE Hard*, with consistency checking for predicate setting instructions and the Traceback PRED optimization, and (3) *Full Hard*, with consistency checking for memory access and predicate setting instructions. We considered the delayed branch optimization for all versions because previous works showed statistically the same fault reduction for both versions. Table 3 summarizes the name of the final fault tolerance techniques and the shows each tested version and optimizations applied.

Table 3 Implemented hardened versions

Label	Application hardening	Move	Traceback MEM	Traceback PRED	Delayed Notification
Unhardened					
SDC Hard	X		X		X
SDC Hard [M]	X	X	X		X
DUE Hard	X			X	X
DUE Hard [M]	X	X		X	X
Full Hard	X				X
Full Hard [M]	X	X			X

We injected 10,000 faults for each combination of (1) case-study application (FFT, Matrix, VectorSum, and Sort), (2) software version (original, SDC Hard, SDC Hard [M], DUE Hard, DUE Hard [M], Full Hard, and Full Hard [M]), (3) GPU configuration (8-, 16-, 32-core), and (4) fault injection location (register file and pipeline registers), reaching 1,440,000 injected faults (VectorSum has only 3 software versions). Each of the 1,44 million faults was injected as a single fault per program execution, meaning that we ran the case-study applications a total of 1,44 million times, each with a single upset. By doing so, our analysis ensures a statistical significance of a 1% error margin with a 99% confidence level [38]. The experiments were performed on Mentor ModelSim running in a workstation with an Intel Xeon CPU @2.5 GHz, equipped with 12 cores and 256 GB of RAM memory, and required about a month.

In the following, we discuss in detail the results for faults injected into the register files and into the pipeline registers.

6.1 Register File

Fig. 4 averages data for the 8-, 16-, and 32-core configurations for the fault injection in the register files, as they presented similar results. It shows, for all four case-study applications and all hardening versions (because the VectorSum does not have control-flow instructions, it does not have a DUE Hard version, and the SDC Hard is the same as the Full Hard), faults classified according to their effects (DUE, SDC, Detected, and Masked).

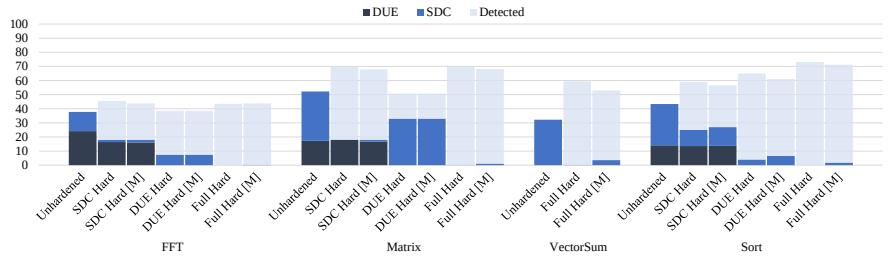


Fig. 4 Fault effects distribution for faults injected into the register files.

When considering SDC effects, SDC Hard and SDC Hard [M] techniques effectively reduced DUE and SDC cases in all applications, showing an average error reduction rate of 88% and 82%, respectively. SDC Hard was able to detect faults more effectively for the data-flow-oriented applications (FFT, Matrix, and VectorSum), followed closely by SDC Hard [M]. On the other hand, both techniques could not detect all SDCs for the FFT application because its control part is larger than in the other two applications. Also, when considering the Sort application, the SDC Hard and SDC Hard [M] showed poor results, being able to reduce SDCs by 61% and 55%, respectively. They could not effectively detect SDCs mainly because the Sort's control-flow includes conditional

instructions in its main loop. Therefore, protecting the memory access instructions alone leaves most of its dynamic instructions unprotected and prone to SDCs. An option to improve the SDC Hard techniques would be not simply targeting memory access instructions. Instead, evaluating all instructions more aggressively, considering their impact in causing SDC effects.

Unlike the memory access techniques, the DUE Hard and DUE hard [M] techniques eliminated all DUEs. Note that DUE Hard and DUE Hard [M] are the same for the data-flow applications because there are no memory access instructions in their control flows. They differ only for the Sort application. Still, for all applications, they detected all DUE effects. Also, note that both techniques were also able to reduce SDC effects for the FFT and the Sort application. Especially when considering the Sort, they achieved better detection capabilities for SDC effects than the SDC Hard techniques. As mentioned previously, the Sort has a conditional comparison instruction that belongs to its control-flow, which can only be hardened by targeting predicate setting instructions. For this application, a predicate setting instruction is far more relevant for SDC effects than the remaining memory access ones.

Finally, when targeting both SDC and DUE effects, one must use the Full Hard technique, followed closely by the Full Hard [M] (with increased SDC effects). As seen for the SDC and DUE Hard techniques, the SDC ones cannot detect DUE effects whatsoever, while the DUE ones cannot detect most SDC effects (except for the Sort, where it reaches 86%). These results show that solely targeting either memory access or predicate setting instructions is not the best option. One should more aggressively select, for each application, which instructions to harden.

6.2 Pipeline Registers

Fig. 5 averages data of the four case-study applications for the fault injection in the pipeline registers, as they produced similar results. It presents faults classified according to their effects (DUE, SDC, and Detected) for all hardening techniques and configurations. Masked effects have been removed for clarity because they represent over 98% of the effects. Although only a small percentage of errors is observed, we can draw tendencies on the fault effects and the software-based hardening techniques detection capabilities.

Considering the original unprotected application, one can notice that SDC effects happen more than DUE ones, up to 2.6 times for the 8-core configuration. As we increase the number of cores, the SDC rate decreases while the DUE effects show a small reduction when moving from the 16- to 32-core configuration. Still, the 32-core configuration shows 1.6 SDCs for each DUE. This reduction happens because the 8- and 16-core configurations put increased pressure on the warp scheduler, resulting in more DUE effects. When we apply the hardening techniques, we achieve a reduction in SDC effects for all techniques, from a 28% reduction for the SDC Hard [M] (32-core) to a 77% reduction for the Full Hard [M] (32-core). Note that the DUE Hard techniques

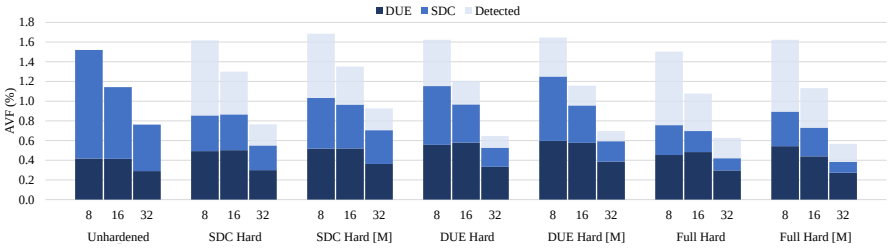


Fig. 5 Fault effects distribution for faults injected into the pipeline registers.

can also detect SDC effects, showing that memory access instructions are not fully correlated with SDC effects. On the other hand, the same cannot be achieved for DUE effects, to the point that, on average, they increase DUEs by 34%. To better understand this behavior's cause, we classified the fault injection location as in the pipeline's datapath or controlpath.

Fig. 6 distributes data from Fig. 5 according the the fault injection location. It shows that fault effects are more common in the controlpath than in the datapath. In the datapath, faults are more easily masked due to a large percentage of unused bits during instruction execution. On the other hand, the controlpath has control bits responsible for the general GPU operation. When affected by a fault, they can more easily propagate the fault while executing any instruction. As these registers are not visible to the user through assembly instructions, our proposed hardening techniques cannot directly target them. Also, by inserting additional instructions, our proposed techniques increase the chance of a DUE effect in the pipeline. To solve this issue, one should consider hardware-based techniques to target specific registers in the pipeline selectively.

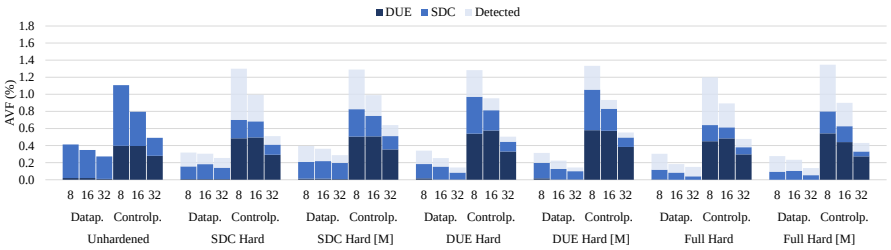


Fig. 6 Fault effects distribution for faults injected into the pipeline registers and classified as datapath (Datap.) or controlpath (Controlp.).

All hardening techniques positively affect the reduction of SDCs in the datapath and controlpath, especially the techniques aimed at reducing SDCs and Full Hard. In the datapath, SDCs occur when a fault propagates to data registers. In the controlpath, SDCs occur mainly when a fault alters the instruction's operation, producing an incorrect value that propagates in the pro-

gram code. In these cases, the faults are detected by our proposed hardening techniques.

7 Design Space Exploration

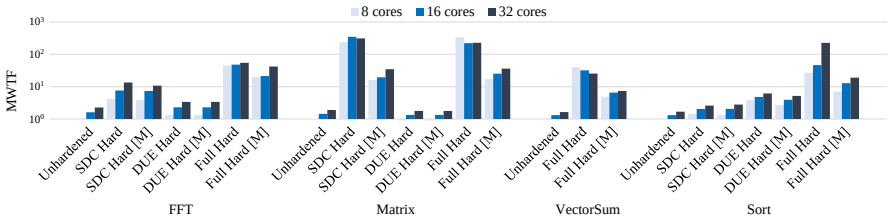
We used the AVF metric to measure and discuss the fault detection capabilities of the proposed software-based hardening techniques in the registers from the pipeline and the register files. The AVF is a useful metric for estimating the probability of failures in the presence of faults for each GPU configuration and the impact of our proposed techniques in reducing this value. On the other hand, the AVF metric fails to account for other essential metrics, such as performance and the number of sensitive bits (i.e., area). Therefore, it is difficult to measure the reliability impact of using different GPU configurations since it directly affects the GPU's performance and area. For example, a 32-core configuration executes the generic application faster than an 8-core configuration, thus increasing reliability but also having more sensitive bits, thus reducing reliability. On top of that, our proposed hardening techniques impact different core configuration performances in different ways.

To account for AVF, performance, and area, we herein adopt a second reliability metric called *Mean Workload to Failure* (MWTF) [39], defined in Eq. 1. A larger MWTF means that more workload can be completed before the system fails. MWTF considers the AVF to a given effect (SDC, DUE, or both), workload execution time (i.e., application's runtime), and the raw error rate (i.e., GPU configuration's raw number of sensitive bits). Because the raw error rate also depends on the circuit technology and environmental conditions, we normalize MWTF over the 8-core configuration running original applications. The normalization process is performed by dividing, for each application, the MWTF of all implemented versions by the MWTF of the unhardened original version running on the 8-core GPU configuration. By doing so, we consider that all configurations are running on the same technology and environmental conditions. Therefore, we remove these factors from the equation, reducing the raw error rate to the number of available registers in each GPU configuration.

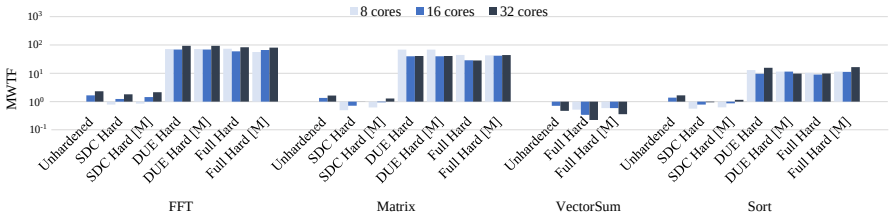
$$\text{MWTF} = (error\ rate \times AVF \times runtime)^{-1} \tag{1}$$

Fig. 7 shows the normalized MWTF, where Figs. 7(a), 7(b), and 7(c) consider as AVF the SDC, DUE, and both effects, respectively. Note that the VectorSum application does not have a Traceback optimization version because it has no control-flow instructions. Thus, it only has a Full Hard and Full Hard [M] hardened versions, which are equivalent to the SDC Hard and SDC Hard [M] versions, respectively.

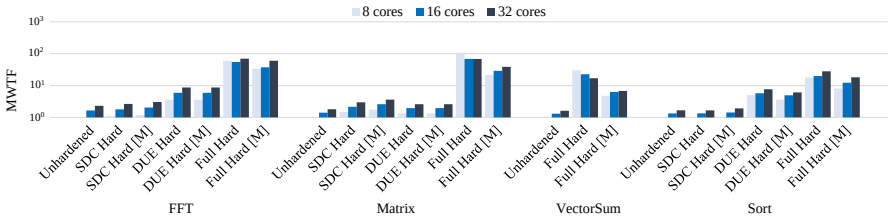
The Mean Workload to SDC Failure results (Fig. 7(a)) show that the use of hardening techniques improves MWTF in all applications, especially the techniques optimized for reducing SDCs. The main reason for this improvement is that techniques can reduce SDC effects in both the register files and the pipeline, thus drastically reducing AVF and improving the MWTF up to 348



(a) Mean Work to SDC Failure



(b) Mean Work to DUE Failure



(c) Mean Work to Failure

Fig. 7 MWTF normalized over original 8-core configuration. (a) Mean Work to SDC Failure. (b) Mean Work to DUE Failure. (c) Mean Work to Failure.

times. In most cases, the increase in the number of cores improves the MWTF, indicating that the execution time is quite relevant. This trend can also be seen in the FFT application running on the 32-core configuration, where the Full Hard [M] presented a better MWTF than the Full Hard, even though it is less effective in reducing errors. In other cases (Matrix with Full Hard and VectorSum with SDC Hard), reducing sensitive bits by reducing cores is the best option.

The Mean Workload to DUE Failure results (Fig. 7(b)) show that the proposed techniques for reducing DUEs improve MWTF by 140 times, on average. On the other hand, SDC Hard and SDC Hard [M] make the applications more sensitive to DUE errors. This trend happens mainly because these hardening techniques do not target DUE effects, and thus their additional instructions increase execution time and sensitivity to DUE effects without helping detect them. An increased execution time makes the pipeline more sensitive to DUE effects due to the additional assembly instructions executed by the GPU. When analyzing the impact of implemented techniques, we can see that the

results vary from application to application. For example, for the FFT, the DUE Hard with the 32-core configuration is the best option. In contrast, for the Matrix, the DUE Hard application running on the 8-core configuration provided better improvement in MWTF. Note that, for all applications, DUE Hard is even more efficient than the Full Hard. This result shows that increasing the number of assembly instructions also increases DUE effects, indicating the high potential advantages of selective protection.

The Mean Workload to Failure results (Fig. 7(c)) show that software-based hardening techniques improve MWTF, achieving up to 106 times improvement for the 8-core configuration on the Matrix application. As with selective protection of DUEs or SDCs, the ideal core configuration setting depends on the application. These results show that MWTF is an interesting metric, especially for applications that need to calculate large workloads and take a long time to complete their tasks. For these cases, the results show that the best fault tolerance technique should consider an optimized balance between its effectiveness in reducing errors, execution time, sensitive bits, and AVF. In this scenario, configurable GPUs can be a good option.

8 Conclusions and Future Work

This work evaluated low-level software-based fault tolerance techniques designed to detect SEU effects in configurable GPU architectures. We adapted and implemented state-of-the-art software-based fault tolerance techniques through low-level assembly code transformations and proposed three optimizations to improve the performance of the hardened case-study applications at costs in reliability, fault detection for specific effects, and host notification time. These novel technique optimizations, called *Traceback*, *Move*, and *Delayed Notification*, were automatically applied to selectively protect three groups of instructions: memory access instructions, predicate setting instructions, and all instructions. To measure the impact of the techniques and optimizations on the configurable GPU, we ran four case-study applications on three GPU configurations. Moreover, we investigated the sensitivity of register files and pipeline registers to radiation-induced faults. A fault injection campaign was performed through simulation at RTL with over 1.4 million faults injected to evaluate the GPGPU's susceptibility to SEUs.

We measured error rate, AVF, and runtime for 216 scenarios, varying hardening techniques, optimizations, core configurations, and case-study applications to explore the design space for hardening the FlexGripPlus configurable GPU architecture. We then presented these data with the MWTF metric, which factors measured data and normalizes them over each original unhardened application running on the 8-core GPU configuration. Our comprehensive design space exploration shows that when equality factoring error rate, AVF, and runtime, the most reliable GPU architecture configuration is not intuitive. Instead, a balance between error reduction, execution time, raw error rate, and AVF must be considered to optimize the GPU architecture to a

given application, environment, or device. Finally, we conclude that navigating the reliability design space is mandatory to improve hardened application efficiency, as simply applying the most expensive software-based technique to the largest available hardware does not guarantee the best reliability.

In the near future, we will address the Traceback optimization and how to create an algorithm for choosing the most effective instructions to harden for a given fault effect. We also intend to develop hardware-based fault tolerance techniques and add them to our design space exploration. Besides, we will consider other reliability metrics. Even though we believe that MWTF is an interesting metric, some designers might favor a variable (e.g., AVF over runtime), thus changing the design space. Finally, we will extend our evaluation to larger more complex applications.

Acknowledgments

This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 722325, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001, Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), and Fundação de Amparo à pesquisa do Estado do RS (FAPERGS).

References

1. Chernikova A, Oprea A, Nita-Rotaru C, Kim B (2019) Are self-driving cars secure? evasion attacks against deep neural networks for steering angle prediction. In: 2019 IEEE Security and Privacy Workshops (SPW), pp 132–137, DOI 10.1109/SPW.2019.00033
2. Hassani R, Aiatullah M, Luksch P (2014) Improving HPC application performance in public cloud. IERI Procedia 10:169–176, DOI https://doi.org/10.1016/j.ieri.2014.09.072
3. Hakobyan G, Yang B (2019) High-performance automotive radar: A review of signal processing algorithms and modulation schemes. IEEE Signal Processing Magazine 36(5):32–44, DOI 10.1109/MSP.2019.2911722
4. Bojarski M, Del Testa D, Dworakowski D, Firner B, Flepp B, Goyal P, Jackel LD, Monfort M, Muller U, Zhang J, et al. (2016) End to end learning for self-driving cars. arXiv preprint arXiv:160407316
5. JEDEC (2006) Measurement and reporting of alpha particle and terrestrial cosmic ray induced soft errors in semiconductor devices. https://www.jedec.org/standards-documents/docs/jesd-89a, accessed: 2021-09-19
6. Oliveira DA, Rech P, Quinn HM, Fairbanks TD, Monroe L, Michalak SE, Anderson-Cook C, Navaux PO, Carro L (2014) Modern GPUs radiation sensitivity evaluation and mitigation through duplication with comparison. IEEE Transactions on Nuclear Science 61(6):3115–3122

7. Pilla LL, Rech P, Silvestri F, Frost C, Navaux POA, Reorda MS, Carro L (2014) Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. IEEE Transactions on Nuclear Science 61(4):1874–1880
8. Slayman C (2010) Soft errors—past history and recent discoveries. In: IEEE International Integrated Reliability Workshop Final Report, pp 25–30
9. Dixit A, Wood A (2011) The impact of new technology on soft error rates. In: International Reliability Physics Symposium, pp 1–7
10. Azambuja JR, Nazar G, Rech P, Carro L, Kastensmidt FL, Fairbanks T, Quinn H (2013) Evaluating neutron induced see in SRAM-based FPGA protected by hardware- and software-based fault tolerant techniques. IEEE Transactions on Nuclear Science 60(6):4243–4250, DOI 10.1109/TNS.2013.2288305
11. Tiwari D, Gupta S, Rogers J, Maxwell D, Rech P, Vazhkudai S, Oliveira D, Londo D, DeBardeleben N, Navaux P, Carro L, Bland A (2015) Understanding GPU errors on large-scale HPC systems and the implications for system design and operation. In: 2015 IEEE International Symposium on High Performance Computer Architecture (HPCA), pp 331–342, DOI 10.1109/HPCA.2015.7056044
12. Hari SKS, Tsai T, Stephenson M, Keckler SW, Emer J (2017) SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp 249–258
13. Gonçalves M, Saquetti M, Kastensmidt F, Azambuja JR (2017) A low-level software-based fault tolerance approach to detect SEUs in GPUs' register files. Microelectronics Reliability 76:665–669
14. Gonçalves M, Saquetti M, Azambuja JR (2018) Evaluating the reliability of a GPU pipeline to SEU and the impacts of software-based and hardware-based fault tolerance techniques. Microelectronics Reliability 88:931–935
15. Mahmoud A, Hari SKS, Sullivan MB, Tsai T, Keckler SW (2018) Optimizing software-directed instruction replication for gpu error detection. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, pp 842–853
16. Rhod EL, Lisbôa CAL, Carro L, Sonza Reorda M, Violante M (2008) Hardware and software transparency in the protection of programs against SEUs and SETs. Journal of Electronic Testing 24(1-3):45–56
17. Condia JER, Du B, Sonza Reorda M, Sterpone L (2020) Flexgripplus: An improved GPGPU model to support reliability analysis. Microelectronics Reliability 109:113660, DOI 10.1016/j.microrel.2020.113660
18. Kadi MA, Janssen B, Yudi J, Huebner M (2018) General-purpose computing with soft GPUs on FPGAs. ACM Transactions on Reconfigurable Technology and Systems 11(1), DOI 10.1145/3173548
19. Goncalves MM, Azambuja JR, Condia JER, Sonza Reorda M, Sterpone L (2020) Evaluating software-based hardening techniques for general-

purpose registers on a GPGPU. In: 2020 IEEE Latin-American Test Symposium (LATS), IEEE, pp 1–6
20. Dimitrov M, Mantor M, Zhou H (2009) Understanding software approaches for GPGPU reliability. In: Proceedings of 2Nd Workshop on General Purpose Processing on Graphics Processing Units, ACM, New York, NY, USA, GPGPU-2, pp 94–104, DOI 10.1145/1513895.1513907
21. Wadden J, Lyashevsky A, Gurumurthi S, Sridharan V, Skadron K (2014) Real-world design and evaluation of compiler-managed GPU redundant multithreading. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), pp 73–84, DOI 10.1109/ISCA.2014.6853227
22. Rech P, Aguiar C, Frost C, Carro L (2013) An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on GPUs. IEEE Transactions on Nuclear Science 60(4):2797–2804
23. Braun C, Halder S, Wunderlich HJ (2014) A-abft: Autonomous algorithm-based fault tolerance for matrix multiplications on graphics processing units. In: 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, IEEE, pp 443–454
24. Pilla LL, Rech P, Silvestri F, Frost C, Navaux POA, Reorda MS, Carro L (2014) Software-based hardening strategies for neutron sensitive FFT algorithms on GPUs. IEEE Transactions on Nuclear Science 61(4):1874–1880
25. Sullivan MB, Hari SKS, Zimmer B, Tsai T, Keckler SW (2018) Swapcodes: Error codes for hardware-software cooperative gpu pipeline error detection. In: 2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, pp 762–774
26. Gonçalves M, Condia JR, Reorda MS, Sterpone L, Azambuja J (2020) Improving GPU register file reliability with a comprehensive ISA extension. Microelectronics Reliability 114:113768, DOI 10.1016/j.microrel.2020.113768
27. Goncalves MM, Lamb IP, Rech P, Brum RM, Azambuja JR (2020) Improving selective fault tolerance in gpu register files by relaxing application accuracy. IEEE Transactions on Nuclear Science 67(7):1573–1580, DOI 10.1109/TNS.2020.2982162
28. Gupta M, Lowell D, Kalamatianos J, Raasch S, Sridharan V, Tullsen D, Gupta R (2017) Compiler techniques to reduce the synchronization overhead of gpu redundant multithreading. In: 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), IEEE, pp 1–6
29. Sundaram A, Aakel A, Lockhart D, Thaker D, Franklin D (2008) Efficient fault tolerance in multi-media applications through selective instruction replication. In: Proceedings of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies, pp 339–346
30. Kalra C, Previlon F, Rubin N, Kaeli D (2020) Armorall: Compiler-based resilience targeting gpu applications. ACM Transactions on Architecture and Code Optimization (TACO) 17(2):1–24

31. Goncalves M, Fernandes F, Lamb I, Rech P, Azambuja JR (2019) Selective fault tolerance for register files of graphics processing units. IEEE Transactions on Nuclear Science 66(7):1449–1456
32. dos Santos FF, Brandalero M, Basso PM, Hubner M, Carro L, Rech P (2020) Reduced-precision dwc for mixed-precision GPUs. In: 2020 IEEE 26th International Symposium on On-Line Testing and Robust System Design (IOLTS), IEEE, pp 1–6
33. Andryc K, Merchant M, Tessier R (2013) Flexgrip: A soft GPGPU for FPGAs. In: 2013 International Conference on Field-Programmable Technology (FPT), pp 230–237, DOI 10.1109/FPT.2013.6718358
34. Lindholm E, Nickolls J, Oberman S, Montrym J (2008) Nvidia tesla: A unified graphics and computing architecture. IEEE Micro 28(2):39–55
35. Oh N, Shirvani PP, McCluskey EJ (2002) Error detection by duplicated instructions in super-scalar processors. IEEE Transactions on Reliability 51(1):63–75
36. Azambuja JR, Lapolli A, Rosa L, Kastensmidt FL (2011) Detecting sees in microprocessors through a non-intrusive hybrid technique. IEEE Transactions on Nuclear Science 58(3):993–1000
37. Mukherjee SS, Weaver C, Emer J, Reinhardt SK, Austin T (2003) A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In: Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., pp 29–40, DOI 10.1109/MICRO.2003.1253181
38. Leveugle R, Calvez A, Maistri P, Vanhauwaert P (2009) Statistical fault injection: Quantified error and confidence. In: 2009 Design, Automation and Test in Europe, IEEE, pp 502–506, DOI 10.1109/DATE.2009.5090716
39. Reis GA, Chang J, Vachharajani N, Mukherjee SS, Rangan R, August DI (2005) Design and evaluation of hybrid fault-detection systems. In: 32nd International Symposium on Computer Architecture (ISCA'05), pp 148–159, DOI 10.1109/ISCA.2005.21